ISI/RR-80-84
October 1980

Neil M. Goldman
David S. Wile

A Database Foundation for Process Specifications

INFORMATION SCIENCES INSTITUTE
4676 Admiralty Way/ Marina del Rey/ California 90291
(213) 822-1511
UNIVERSITY OF SOUTHERN CALIFORNIA

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: ISI/RR-80-84	2. GOVT ACCESSION NO.: AD-A092 567
3. RECIPIENT'S CATALOG NUMBER:	
4. TITLE (and Subtitle): A Database Foundation for Process Specifications •	5. TYPE OF REPORT & PERIOD COVERED: Research rept.
6. PERFORMING ORG. REPORT NUMBER:	
7. AUTHOR(s): Neil M. Goldman, David S. Wile	8. CONTRACT OR GRANT NUMBER(s): DAHC15-72-C-0308
9. PERFORMING ORGANIZATION NAME AND ADDRESS: USC/Information Sciences Institute, 4676 Admiralty Way, Marina del Rey, CA 90291	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: ARPA Order-2223
11. CONTROLLING OFFICE NAME AND ADDRESS: Defense Advanced Research Projects Agency, 1400 Wilson Blvd., Arlington, VA 22209	12. REPORT DATE: October 1980
	13. NUMBER OF PAGES: 33
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): -------	15. SECURITY CLASS. (of this report): Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

This document is approved for public release and sale; distribution is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

----

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

abstract data model, formal process specification language, formal specification, natural language, specification language

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

(OVER)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

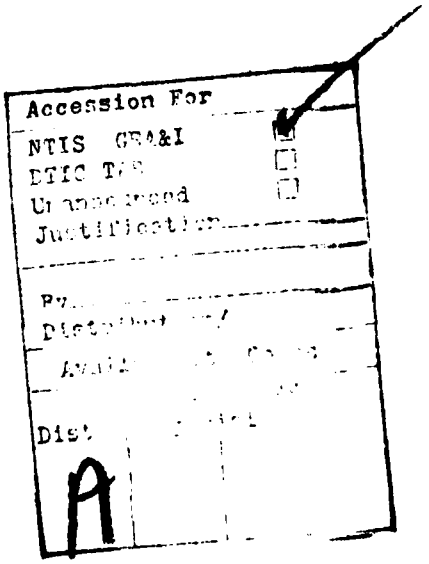
UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

AD A092952

20. ABSTRACT

A language suitable for system specification should allow a specification to be based on a cognitive model of the process being described. In part, such a language can be obtained by properly combining certain conceptual abstractions of data models with reference and control concepts designed for programming languages. Augmenting the resulting language with formal versions of several natural language constructs further decreases the cognitive distance between specifications of large systems and the modelled world.

Several core elements of such a specification language are developed in this report. Emphasis is placed on modes of expression, such as declarative constraints and temporal reference, which are derived from natural language but are not available in existing formal languages.



ISI/RR–80–84
October 1980

Neil M. Goldman
David S. Wile

A Database Foundation for Process Specifications

INFORMATION SCIENCES INSTITUTE

UNIVERSITY OF SOUTHERN CALIFORNIA

4676 Admiralty Way/Marina del Rey/California 90291
(213) 822-1511

THIS RESEARCH IS SUPPORTED BY THE DEFENSE ADVANCED RESEARCH PROJECTS AGENCY UNDER CONTRACT NO. DAHC15 72 C 0308, ARPA ORDER NO. 2223.
VIEWS AND CONCLUSIONS CONTAINED IN THIS REPORT ARE THE AUTHORS' AND SHOULD NOT BE INTERPRETED AS REPRESENTING THE OFFICIAL OPINION OR POLICY OF DARPA, THE U.S. GOVERNMENT, OR ANY PERSON OR AGENCY CONNECTED WITH THEM.

CONTENTS

Acknowledgments iv

1. Introduction *1*

2. Specifying the Domain of a Process *2*
   - 2.1 Objects and Types *3*
   - 2.2 Relations *4*
   - 2.3 Expressions, Patterns, and Predicates *5*
   - 2.4 Constraints *8*
   - 2.5 Derived Relationships *8*

3. Specifying the Dynamics of a Process *10*
   - 3.1 Control Structures *11*
   - 3.2 Actions *13*
   - 3.3 Procedural Requirements *14*
   - 3.4 Data Triggered Processing *15*
   - 3.5 Temporal Reference *17*
   - 3.6 Process Granularity *19*
   - 3.7 Constraints and Non-Determinism *21*
   - 3.8 Anomaly Control *24*

4. Conclusion *24*

References 27

ACKNOWLEDGMENTS

The ideas presented in this report arose during lengthy discussions within a closely knit group of colleagues. As such, the individual contributions of the group members are impossible to identify. The authors would like to acknowledge the major influence on this work of other members of this group, consisting of Bob Balzer, Lee Erman, Martin Feather, and Phil London.

I. INTRODUCTION

A major effort is under way within computer science to design new languages that will enhance the development of reliable and maintainable software, particularly for large applications. Through careful structuring [13] and encapsulation [17] of information, some new *programming* languages permit hierarchical development of large programs. Each layer of the hierarchy is understandable in terms of properties abstracted from modules in the lower layers. From the definitions of the modules at the base of the hierarchy, a compiler can (or could) produce an acceptable implementation of the entire system.

Another line of development has been a search for languages with more expressive power than is provided in programming languages [6, 9, 16]. The designers of these *specification* languages are willing to forego the ability to have their specifications mechanically compilable into (efficient) implementations. In return, they hope to make it easier to write a formal specification of a process and, more important, to increase the likelihood that the process specified is indeed the one desired.[1]

Balzer and Goldman [2] enumerate several language principles claimed to be beneficial for both the creation and maintenance of large software systems. These include the requirement that a specification be a *cognitive model* of the process being specified. In general, a software system is intended to represent the activity in some "ideal world," which may be an abstraction of a real-world process, a purely mental conception of the desired behavior, or a combination of the two. We hope to minimize the "translation distance" from this ideal world to its formal representation by permitting that representation to model directly the ideal insofar as possible. This should increase our confidence that a specification in fact matches the intended ideal.

Maintenance involves, as its first step, translating changes to the ideal world into corresponding changes to the specification. If the specification is a cognitive model, the amount of change required in the specification should be comparable to the amount of change in the ideal. We believe that most maintenance changes represent fairly small changes to the ideal world.

A good source of ideas for language components that help in constructing cognitive models is *natural language*. Natural language has been roundly criticized in some circles [12] because of its informality and ambiguity. Clearly a formal language cannot adopt these characteristics, although they contribute significantly to the utility of natural languages for communication. But natural languages also contain a variety of modes of expression that are richer than those provided by even the highest level programming languages, yet that have readily formalizable counterparts. A number of these are developed in this report. One reason for their absence from programming languages is undoubtedly the difficulty of providing for (efficient) computer implementation of their

---

[1] We think of our language as specification oriented. Our goal is to have programs produced from specifications through a transformational development [3].

general use. This is not a restriction on natural languages, which are generally concerned with communicating only the requisite external behavior of a process. The implementation of that process, whether on a computer or otherwise, is an orthogonal concern.

Since the pioneering work of Codd [8] on relational data bases, several distinct data models have been developed and studied. An often noted characteristic of these models is that they provide not only the basis for machine storage and manipulation of data, but a cognitive model of the data domain as well. In fact, these data models bear great similarity to the *semantic nets* used in artificial intelligence programs for understanding natural language, as demonstrated in [15].

The specification language described below is based on such a data model. We believe that any process can, and should, be defined in terms of a variety of entity *types*, specific to the process, which are associated with one another by means of process specific *relations*, and acted upon by process specific *actions*. These actions consist of combinations of creation and destruction operations on these entities and associations.

Section 2 of this report develops the *static* aspects of this model. It presents means for specifying the structural regularity of the data domain, including a hierarchy of object types, relations on those types, constraints on data states, and *derived* relationships (alternative "views"). It also lays out a powerful query language for expressing predicates on the data states and for referring to objects in those states. Section 3 introduces the means for defining the *dynamic* aspects of a process. These mechanisms rely on the underlying data model to define a number of rich constructs not available in even very high-level programming languages. We point out how each of these corresponds to a descriptive capability in natural language, and why each enhances the specification of large systems.

Notation

In this report, meta-concepts of the language are printed within angle brackets (<>). In syntactic templates, braces ({}) enclose optional elements, and ellipses (...) indicate allowable repetition of the preceding constituent. The "reserved words" of the language are underlined.

The report uses examples drawn from an ideal world of ships, ports, piers, cargos, etc. Within the examples and text describing them, the names of these "types" are printed in **bold lower case**. Variable and parameter names are printed in *italicized lower case*. The names of relations and actions are printed in **BOLD UPPER CASE**. Finally, objects referred to literally are printed in *Mixed Case Italics*.

2. SPECIFYING THE DOMAIN OF A PROCESS

An ideal world is not an arbitrary collection of objects related in unstructured ways. Rather, the objects can be categorized into various *type* classifications. There are only

certain kinds of relationships in which the various types of objects may participate. Neither does the ideal world permit arbitrary combinations of these objects and relationships to coexist.

It is important to capture the structure of the ideal world in the specification. Doing this actually makes it easier to specify the process taking place in the ideal world. Even more important, it enhances our ability to alter the specification so that it conforms to a changed ideal world. This is the source of our ability to *maintain* software systems created from the specification. The structure of the ideal world is specified through a variety of ⟨*declaration*⟩ forms described in the following sections.

2.1 Objects and Types

The various types of objects in the ideal world are named in *type declarations*. The simplest type declaration simply lists the names of various types:

```
type ship; pier; cargo; slip; crewmember end type
```

A name so declared may be used as a ⟨type identifier⟩ elsewhere. Some types may be subtypes of others; this is declared by including a modifier in a type declaration:

```
type oiltanker, a kind of ship :
     officer, a kind of crewmember
end type
```

This declaration states that every **oiltanker** is also a **ship**. Analogously, it makes **officer** a subtype of **crewmember**. Although the collection of **oiltankers** and **ships** may change as a process executes, no object is ever an **oiltanker** but not a **ship**. There is no need for the specification to include manipulations of the data specifically to maintain this invariant; it is ensured by the declaration.

Smith and Smith [14] have pointed out many of the virtues of having such type hierarchies from the standpoint of database design. The most salient advantages in a specification language are that any relations and operations defined on a type are automatically defined on its subtypes, and that the types can be used in the data manipulation language to strengthen predicates in a natural and concise manner.

We can also define synonyms for types, and define one type as a restriction of another:

```
type   message, = string;
       latitude, = integer in range [-90,90];
       longitude, = integer in range [-180,180]
end type
```

This declaration states that **message** is a synonym for the predefined type **string**, and that **latitude** and **longitude** are particular subranges of the predefined type **integer**.

Sometimes a specification must refer to particular individual objects. These can be introduced when their types are declared.

type **port**, ⊃ {*Seattle*, *Santa Barbara*} end type

declares that the type **port** has (at least) two distinct instances, which will be referred to in the specification by the literals *Seattle* and *Santa Barbara*, whereas

type **grain**, a kind of **cargo**, = {*Corn*, *Wheat*} ;
fuel, a kind of **cargo**, = {*Oil*, *Natural Gas*}
end type

declares **grain** and **fuel** to be subtypes of **cargo**, with their instances totally enumerated by literals. No further instances of **grain** or **fuel** may be defined or created.

2.2 Relations

Our conception of *relations* corresponds closely to that seen in Chen's entity-relationship diagrams [7].[2] A relation is defined to have some number of *roles*, each role having a name and a type. At any stage of a process, each relation contains a collection of *tuples*. Each tuple in a relation has an object *filling* each of its roles. The object filling a role must be an instance of that role's type. The role types thus serve to restrict the tuples that can appear in a relation.

The declaration of an *n-ary* relation has the form:

relation <relation identifier> (<role>$_1$,...,<role>$_n$);
...
end relation

Each <role> is denoted by an <id:type>, which is simply an arbitrary name, followed by a colon, followed by a type identifier, e.g., *s*:**ship**. The identifier preceding the : is the role name, and the type identifier names the role type. By convention, a type name *t* alone can be used as an id:type to abbreviate *t*:**t**, and a name of the form *t.d*, for any digit d, in place of *t.d*:**t**. For example, *ship*:**ship** can be abbreviated as *ship*, and *ship.1*:**ship** as *ship.1*.

A relation **PORTOFCALL** between **ships** and **ports** for which they are bound and a relation **SHIPPINGPIER** between **piers** and the **cargos** that they handle are declared by:

relation
PORTOFCALL(*ship*, *port*);
SHIPPINGPIER(*cargo*, *pier*)
end relation

Unless otherwise specified, a relation is many-to-many (to-many ...). The ideal world relationships being modeled by **PORTOFCALL** and **SHIPPINGPIER** are both many-to-many.

---

[2] Unlike Chen, we do not distinguish between inter-entity relationships and values of attributes of entities. We believe this distinction belongs in the realm of implementation, not specification, being based on the usage of information rather than the nature of the information itself.

The concept of a *key* of a relation is familiar in relational data bases, and is important to capture in a specification. One or more keys for a relation can be specified by a modifier on the relation declaration. Each key consists of one or more role names. Another important concept we call *covering*. A relation covers a role if every object of that role's type fills that role in at least one tuple in the relation. If a relation covers a role that is a key of the relation, then every object of that role's type fills the role in *exactly* one tuple in the relation. In this case, we say the relation *defines* the role.[3]

```
relation
    CAPACITY(ship,volume), defines ship;
    CONTAINS(ship,cargo,volume), key is (ship,cargo);
    PIERPORT(pier,port), defines pier, covers port;
    SLIPS(pier,slip), defines slip, covers pier;
    BERTH(ship,slip), key is ship, slip
end relation
```

These declarations specify that every **ship** has a single **volume** as its **CAPACITY**, that **ships CONTAIN volumes of cargo**, but a given **ship** has only a single **volume** of a given **cargo** at any time, every **pier** is in a particular **port** and every **port** has at least one **pier**, every **slip** is at a particular **pier** and every **pier** has at least one **slip**, and that **BERTH** relates subsets of **ships** and **slips** in a one-to-one correspondence. Just as a role's type restricts the individual tuples in a relation, a relation's keys and coverings restrict the collection of tuples in the relation.

2.3 Expressions, Patterns, and Predicates

An <expression> is a constituent of the language that is used to *refer* to objects. The simplest expression is a *literal*, such as *5000* or *Corn*, which refers to the same object wherever it is used in a specification. The *referent* of a literal is fixed for all time.

A <variable> is a name that may be used as an expression. The referent of a variable may change from one use to another. Each variable in the language is declared as the identifier in a :type, and the referent of the variable must always be an instance of the type that appeared in its declaration.

Expressions may be combined by operators and function names, as in a conventional programming language, to produce other expressions. But any expression, no matter how complex, is only a means for referring to an object; it does not specify any activity that changes objects or relationships.

A <pattern> has the form

<relation identifier>(<expression>$_1$, ..., <expression>$_n$)

where the named relation is n-ary. A pattern *matches* a tuple if each object filling a role

[3] It is occasionally the case that a role serves as a key for some subtype of its type, but not for the entire type. Similarly, a relation may cover a role for some subtype of the role's type. It is possible to succinctly declare key, covering, and defining roles for a subtype of the role's type, but our examples will not require the capability.

in the tuple is the referent of the corresponding expression.[4] The correspondence is the natural positional correspondence between expressions in the pattern and roles in the relation declaration.

A pattern may be used as a <predicate>. The pattern is said to be *True*, or to *hold*, in a particular data state if the named relation contains any tuple that matches the pattern; otherwise it is said to be *False* in that state. Also,

<expression> = <expression>

is a predicate that holds if and only if the expressions have a common referent. Finally,

<expression> isa <type identifier>

holds if the (some) referent of <expression> is an instance of the named type. Predicates may be combined with the logical operators ∧, ∨, ¬, and => with the traditional meanings.

Predicates may also be written with *quantified* variables. ∀<id:type> <predicate> holds in a given data state if <predicate> holds in that state for every assignment of an existing object of the quantified variable's type to that variable. ∃<id:type> <predicate> holds if there exists any such assignment for which <predicate> holds. For example,

$$\exists s{:}\ \textbf{ship}, v{:}\ \textbf{volume}\,(\textbf{CONTAINS}(s, \textit{Corn}, v) \wedge \textbf{PORTOFCALL}(s, \textit{Seattle}) \wedge v \geq \textit{20k-Cubic-Meters}$$ [5]

holds if there exists some **ship** bound for *Seattle* and some **volume** of *Corn* of at least *20k-Cubic-Meters* on that **ship**. We say that the predicate holds *for the assignment* of that **ship** and **volume** to the variables *s* and *v*, respectively.

A predicate that would test for the existence of any **oiltanker** bound for *Santa Barbara* could be written:

$$\exists ship\ (\textbf{PORTOFCALL}(ship, \textit{Santa Barbara}) \wedge ship\ \underline{\text{isa}}\ \textbf{oiltanker})$$

or more naturally as

$$\exists oiltanker\ \textbf{PORTOFCALL}(oiltanker, \textit{Santa Barbara})$$

which might hold for several distinct assignments of **oiltankers** to the variable *oiltanker*.

English *noun phrases* are a very rich form of expression. They provide the power to refer to objects by describing them; i.e., by predicating how they relate to other, possibly also described, objects: e.g., "a ship containing at least 20000m^3 of corn and bound for Seattle." Through the use of possessive and reflexive pronouns, the descriptions can even refer to the object being described: "an employee who manages

---

[4] As we shall see, some expressions may be non-deterministic, having multiple referents. A pattern matches a tuple provided the objects in the tuple are among the referents of the corresponding expressions.

[5] Actually, the predicate ≥ could not be used to compare objects of type **volume** unless an ordering on **volumes** was defined. Such orderings are not covered in this report.

himself."[6] This richness is available in formalized form through use of the *predicate-based expression*:

[<id:type> | <predicate>]

where <predicate> may use the variable name in <id:type> freely. If ∃<id:type> <predicate> holds for some assignment of an object to that variable, then that object is a referent of the expression. Such expressions, like their English counterparts, may be *non-deterministic*, having many referents, *deterministic*, having exactly one referent, or *anomalous*, having no referents.[7] Formally, "a ship containing at least 20000m^3 of corn and bound for Seattle" is expressed:

[*s*:**ship**|**CONTAINS**(*s*,*Corn*,[*v*:**volume**|*v*≥*20k-Cubic-Meters*]) ∧ **PORTOFCALL**(*s*,*Seattle*)] (1)

The expression [<id:type>|True], which refers to any instance of some type, may be abbreviated as [<type identifier>]. It is common for such expressions to appear in a pattern with the type identifier naming the type of the role in which the expression appears. In that case, the expression may simply be written as the symbol **$**. Thus,

PORTOFCALL([**oiltanker**], *Santa Barbara*)

will match tuples in the **PORTOFCALL** relation having *Santa Barbara* in the *port* role and any **oiltanker** in the *ship* role.[8] The more general pattern

PORTOFCALL([**ship**], *Santa Barbara*)

would allow a match for any **ship**, not just an **oiltanker**, and could be written simply as:

PORTOFCALL(**$**, *Santa Barbara*)

It is also common to find predicate-based expressions in which all uses of the distinguished variable in the predicate are in roles of the same type as the variable. In this case the predicate itself, written with the symbol * replacing the variable, may be used as an expression. For instance,

CONTAINS(*, *Corn*, [*v*: **volume** | *v*≥*50tons*]) ∧ **PORTOFCALL**(*, *Seattle*)

is equivalent to (1) above.

---

[6] The noun phrase also derives power from its informality. While we sometimes use a fairly explicit verb to indicate a relation -- "the captain *serving* on the ship" -- it is more common to condense the relation to a vague preposition -- "the captain of the ship" -- or to simply provide a syntactic indication that some relationship exists -- "the ship's captain."

[7] The collection of referents depends on the collection of tuples in the data base, and on the objects assigned to any variables used freely within the predicate.

[8] This is distinct from the pattern **PORTOFCALL**(oiltanker,Santa Barbara), which uses oiltanker freely. This pattern would only match the tuple for the specific oiltanker that was the referent of oiltanker.

2.4 Constraints

We have seen how role types and relation keys serve to constrain the tuples and tuple collections that can coexist in a relation. There may also be constraints in the ideal world which correspond to tuples and collections of tuples that may not coexist in the data base as a whole. A declaration of the form:

constraint <predicate>; ... <predicate> end constraint

outlaws any data state in which any of the predicates holds. The constraint

constraint $\exists$*oiltanker* **PORTOFCALL**(*oiltanker*, *Santa Barbara*) end constraint

prohibits any **oiltanker** from ever having *Santa Barbara* as a destination. A second constraint,

constraint $\exists s$: **ship**(**CONTAINS**(s, [fuel], $) $\wedge$ **CONTAINS**(s, [grain], $)) end constraint

prohibits the mixing of fuel with grain in a ship at any one time. Thus, for instance, a ship could not contain both *Oil* and *Corn*.

The essential "meaning" of the **CAPACITY** relation comes from its appearance in a constraint:

constraint $\exists s$: **ship**($\sum$ {**CONTAINS**(s, $, $\star$)} > **CAPACITY**(s, $\star$)) end constraint

which prohibits the sum of volumes of various cargos contained in a ship from exceeding the capacity of the ship.

Constraints restrict the data states that a process may legitimately create. They play a far more central role in the specification language than they do in database languages. That role is described in section 3.7 below.

2.5 Derived Relationships

It is convenient to be able to refer to relationships that are derived from others. For instance, a port "handles" *Oil* if it has a pier at which *Oil* can be loaded and unloaded. It is important to be able to define the "handles" relation in such terms and to use it in patterns in the same way as any other relation. It is unacceptable for the relationship to be given an independent definition and manipulated by the specified process in such a way as to explicitly maintain its invariant connection to other relations. This invariant should be declared explicitly and its maintenance ensured by that declaration.

These invariants are defined by giving the relation a normal declaration, including roles and keys, and using it in a derivation as well.

```
derivation
  <derivation name>(<id:type>1,...,<id:type>n)
  antecedent <predicate>
  consequent <pattern> ;
   ...
end derivation
```

In any process state for which

$$\exists \langle id{:}type \rangle_1, \ldots, \langle id{:}type \rangle_n \ \langle predicate \rangle$$

holds for some assignment to the variables $\langle id{:}type \rangle_i$, the tuple corresponding to <pattern> for that assignment is taken as being *implicit* in the data base. No distinction is made in the language between implicit and explicit relationships.[9]

The only variables that may appear freely in <predicate> or <pattern> are the $\langle id{:}type \rangle_i$. Each expression in <pattern> must be deterministic. This ensures that the tuple corresponding to <pattern> for any particular assignment to the variables is well defined.

Derivations can be used to define the relationships

- A ship is moored at a pier.

- A ship is in a port.

- A port handles a cargo.

```
relation MOORAGE(ship, pier), key is ship;
         INPORT(ship, port), key is ship;
         HANDLES(port, cargo)
end relation
derivation
DMOOR(ship, pier, slip)
antecedent BERTH(ship, slip) ∧ SLIPS(pier, slip)
consequent MOORAGE(ship, pier);

DINP(ship, port, pier)
antecedent MOORAGE(ship, pier) ∧ PIERPORT(pier, port)
consequent INPORT(ship, port);

DHAND(cargo, port, pier)
antecedent SHIPPINGPIER(cargo, pier) ∧ PIERPORT(pier, port)
consequent HANDLES(port, cargo)
end derivation
```

Note that **MOORAGE** is given a derivation in terms of **BERTH** and **SLIPS**, and is itself used in the derivation of **INPORT**. It is acceptable for a relation to be given several independent derivations. The existence of a derivation rule for a relation does not prohibit the direct insertion of tuples in that relation by the specification. For instance, when arriving at a port, there may be a time when the INPORT relationship holds for that ship before it ever is positioned in a slip at a pier.

---

[9] The only exception to this concerns deletion of tuples. Any attempt to delete a tuple that would still exist implicitly following the deletion is considered anomalous.

3. SPECIFYING THE DYNAMICS OF A PROCESS

The purpose of writing a specification is to describe formally the behavior that takes place in the ideal world. The essence of this behavior is the sequential change in the collection of objects and associations. A specification language <statement> is used to define a transition from one such state to another. The transitions are ultimately composed of five basic data transitions:

- Object Creation -- Seldom are the literal objects named in the static domain model the only objects that exist in the ideal world. New piers, ships, and even ports may come into existence as part of the process. The creation of a new object is specified by the statement:

  create <type>

  This specifies the creation of an entirely new instance of <type>, distinct from all objects currently (or previously) existing.

- Object Destruction -- The ideal world need not be cumulative. Sometimes objects cease to exist. The statement

  destroy <expression>

  specifies the destruction of <expression>'s referent and of all tuples in which that referent appears.

- Tuple Insertion -- New associations are created by the statement:

  insert <pattern>

  which will add to the data base a new tuple matching <pattern>. If the tuple to be added already holds, the insert operation causes no change.

- Tuple Deletion -- Associations are removed by the statement:

  delete <pattern>

  which will remove from the data base a tuple matching <pattern>. If no tuple in the database matches <pattern>, the delete operation causes no change.

- Tuple Update -- A change of the object filling a particular role in a tuple is specified by:

  update <role-name> in <pattern> to <expression>

  which changes the object filling the indicated role in some tuple in the database matching <pattern> to the referent of <expression>. More precisely, the semantics of update are those of a delete followed by an

insert,[10] treated as a single database change. The symbol **oldvalue** may be used in <expression> to reference the object originally filling the role being updated.

All of these statements, with the exception of create, may be non-deterministic. That is, due to the appearance of non-deterministic expressions within the statements, there may be distinct changes to the data base, each of which meets the semantic requirements of the statement. It is occasionally desirable to make a change involving not just one of the objects specified non-deterministically, but all of them. This can be specified with statements destroyall, insertall, deleteall, and updateall.[11] Thus, the salary of every officer could be increased by 5 percent via:

updateall *salary* in **SAL([officer],$)** to 1.05*oldvalue

3.1 Control Structures

To specify a process, it must be possible to state under what conditions and in what order various data transitions take place. *Control structures* are the means for accomplishing this. The control structures available in most high-level programming languages are also useful in specifications. In this report, the only unconventional control structure introduced is the *demon* (see section 3.4). Otherwise, we will confine ourselves to sequencing, conditionals, and iteration.

Sequencing is indicated by separating successive <statement>s by semicolons:

```
<statement>; ... <statement>
```

To meet the syntactic requirements of the language, it is often necessary to bracket a sequence of <statement>s so that it may be used as a single <statement>:

```
begin<statement>; ... <statement> end
```

In mathematics, we are familiar with problem descriptions that include statements such as "Let x, y, and z be numbers such that P(x,y,z). Then ..." This provides a way of introducing some new names, specifying, or restricting, the values to which they refer, and then using those names in further statements. This facility is provided for with the syntax:

```
begin
     ∃<id:type>,...,<id:type> <predicate>;
     <statement>; ... <statement>
end
```

If <predicate> holds for some assignment to the variables, then the variable environment

---

[10] This gives update a meaning both when no tuple matches <pattern> and when the altered tuple is identical to an existing one.

[11] These are not simply iterations making a single transition on each loop, but are *primitive* transitions, as described in section 3.6.

in effect outside this block is extended accordingly. The <statement>s are then executed sequentially in the extended environment. Since the <predicate> may be true for many distinct extensions, the block may be non-deterministic. If there is no assignment satisfying <predicate>, the block is anomalous.[12]

Conditionality is expressed by a <statement> with the conventional syntax:

if <predicate> [then <statement>$_1$] [else <statement>$_2$]

which has the meaning of <statement>$_1$ if <predicate> holds and of <statement>$_2$ otherwise. A conditional <expression> is specified analogously:

if <predicate> then <expression>$_1$ else <expression>$_2$

Another useful capability is to have a conditional <statement> in which the predicate contains existentially quantified variables, permitting the "then" clause to refer to the assignment that satisfied the predicate. In a conditional <statement> or <expression> having a predicate of the form:

∃<variable>,...,<variable> <predicate>

the variable environment surrounding the conditional is extended for the "then" clause to incorporate the portion of the assignment satisfying the predicate for the existentially quantified variables. The variable environment for the "else" clause is that in effect around the conditional itself. For example, the formal representation of "If there is a ship in Santa Barbara containing 20000m^3 of grain, schedule it to stop in Seattle" is:

```
if ∃ship (INPORT(ship, Santa Barbara) ∧
            CONTAINS(ship, [grain], [v: volume | v ≥ 20k-Cubic-Meters]))
  then insert PORTOFCALL(ship, Seattle)
```

Finally, a simple but powerful form of iteration consists of doing the same activity in *every* variable assignment for which some predicate holds:

whenever <predicate> do <statement>

specifies doing <statement> in *every* extended assignment for which <predicate> holds. The extensions are determined, as in the case of conditionals, by the leading existentially quantified variables on <predicate>. The order of assignments in which <statement> is to be done is non-deterministic. All are calculated with respect to the data state existing prior to the initiation of the iteration; the effects of <statement> in one environment have no bearing on the collection of assignments used. The example above may be rewritten to send all ships with 20000m^3 of grain to Seattle:

```
whenever
  ∃ship (INPORT(ship, Santa Barbara) ∧
            CONTAINS(ship, [grain], [v: volume | v ≥ 20k-Cubic-Meters]))
do insert PORTOFCALL(ship, Seattle);
```

[12] One of the powers of English is that it rarely forces us to introduce "variable" names like x, y, and z. Unfortunately, that power appears to derive in large part from the informal aspect of the language.

3.2 Actions

The basic specifier-defined building blocks of a specification are *actions*. They are the analog of procedures in a program. An action is a parameterized <statement> that expresses some data transition, or sequence of transitions, in terms of its parameters. The <statement>, or *definition*, specifies the transition(s) through the use of the primitive database operations and *invocations* of other actions. An action is declared by:

```
action
   <action identifier> (<formal parameter> ... <formal parameter>),
      definition <statement>;
   ...
end action
```

The <action identifier> provides a name for the action. Each <formal parameter> is simply an <id:type>. The variable environment for <statement> consists of the <formal parameter>s, with an assignment in which each parameter refers to the object used as the corresponding actual parameter in an invocation.

An invocation is denoted by:

```
<action identifier>(<expression>, ..., <expression>)
```

with the usual positional correspondence of <expression>s in the invocation to the parameters of the action. The referents of the <expression>s in the invocation environment become the referents of the corresponding parameters of the action within its definition. If any of the <expression>s is anomalous, then the invocation is anomalous.

Example

One activity in shipping is the loading of a given volume of some cargo onto a ship. This action would be declared by:

```
action
   LOADSHIP(ship,cargo,incr:volume),
     definition
      if CONTAINS(ship,cargo,$) then
         update volume in CONTAINS(ship,cargo,$) to oldvalue+incr
        else insert CONTAINS(ship,cargo,incr)
end action
```

The definition of **LOADSHIP** is simply to increment the volume of *cargo* contained by *ship* by an amount *incr*, or to insert a new **CONTAINS** tuple if *ship* did not contain any *cargo* yet.[13]

---

[13] By specifying a *default* volume for **CONTAINS** to be a literal *0volume* (the additive identity for volumes), the conditional could be replaced by its "then" clause. Defaults are not discussed in this report, but are a valuable specification mechanism. Also, the operator + could not be used with objects of type volume unless given a suitable definition in the specification.

Another activity is the movement of a ship to a pier, which must happen when the ship is to be loaded or unloaded. We shall declare the action by:

```
action
    MOVESHIP(ship, pier),
        definition
          if MOORAGE(ship, pier) then
                comment no movement needed end comment else
                update slip in BERTH(ship, $) to SLIPS(pier, *)
end action
```

If the ship is already at the specified pier, then **MOVESHIP** does nothing. Otherwise, it updates the **BERTH** tuple for the ship to indicate that it is at some slip at the desired pier. This slip is specified with the non-deterministic pattern expression **SLIPS**(*pier*,*).

3.3 Procedural Requirements

In the ideal world, there is not only regularity in the state of data relationships, but in the realm of processing as well. The regularity in the types of objects on which a given action is performed is captured by the typing of the formal parameters of an action. At any point in a process, it may be the case that the data and variable assignments must satisfy some predicate for the execution to be feasible. This can be specified by including <requirement>s at appropriate points in the control structure:

```
require <predicate>
```

A require declaration can appear wherever a statement can appear. It signifies that the predicate must be true at the point in execution where it appears.

Two common points to include <requirement>s in a specification are at the initiation and completion of actions. These have been singled out syntactically and may be declared as preconditions and postconditions of an action, rather than included within the action's definition.[14]

It is sometimes desirable to state a requirement on the transition achieved by an action, rather than (or in addition to) requirements on the initial and final states. This can be done with a syntactic means in the postcondition. Any expression or predicate in a postcondition preceded by the marker old refers to its value or truth in the data state at action initiation, rather than termination.[15]

Example

Suppose the action **MOVESHIP** defined above can, in the ideal world, only be used to move a ship to a pier if the ship is already in the port containing that pier. This fact is

---

[14] Preconditions and postconditions may refer to the action's operands by using the formal parameter names as free variables.

[15] Section 3.5 describes a more general facility for reference to past data states.

captured by placing a precondition on the action:

```
action
   MOVESHIP(ship, pier),
       precondition INPORT(ship, PIERPORT(pier, ☆)),
       definition ...
end action
```

The requirement that cargo can only be loaded onto a ship if it is moored at a pier which handles that cargo can be captured by a precondition on the action **LOADSHIP**:

```
action
   LOADSHIP(ship, cargo, incr: volume),
       precondition ∃pier SHIPPINGPIER(cargo, pier) ∧ MOORAGE(ship, pier),
       definition ...
end action
```

Occasionally it is necessary for a port to deal with a new class of cargo. This may necessitate a major shakeup in the assignment of cargos to piers in that port. The action that makes the necessary changes might need to take many factors into account, and might well be expressed best with some non-determinism, since several reassignments might be equally acceptable. However, it might be absolutely unacceptable to deassign a *Natural Gas* pier (due to excessive costs or government regulations). Also, the reassignment must still leave all originally handled cargos still handled, though not necessarily at the same pier.

```
action
   ADDCARGO(cargo, port),
     precondition  ¬ HANDLES(port, cargo),
     definition ... ,
     postcondition  ∀pier(old SHIPPINGPIER(Natural Gas, pier) =>
                          SHIPPINGPIER(Natural Gas, pier)) ,
     postcondition ∀cargo(old HANDLES(port, cargo) => HANDLES(port, cargo))
end action
```

3.4 Data Triggered Processing

In describing a process, it is convenient to be able to make statements of the form "whenever <trigger> is the case, do <response>". <trigger> is some condition on the objects being manipulated by the process and, perhaps, on the control state of the process as well. <response> is itself a process to be performed when that condition is met.

Since all information about the objects is captured in the data base, the predicate language provides a natural formalism for expressing those triggering conditions dependent on object associations and types.

Such *demons* have been permitted in various AI languages [4, 5]. Since these are programming languages, rather than specification languages, they have severely restricted the expressive power permitted in the trigger condition. As a result,

computations triggered by complex conditions have to be "programmed" in these languages, spreading pieces of the condition throughout the program. In a specification, however, the full power of the predicate language, including typed variables, logical operators, and quantification, can be permitted in the trigger without sacrificing any desirable specification properties.

Syntactically, <demon>s may be declared:

```
demon
    <demon identifier>(<demon parameter> ... <demon parameter>),
    trigger <predicate>,
    response <statement>;
    ...
end demon;
```

The <demon identifer> becomes the name of the demon. Each <demon parameter> is an <id:type>. A demon specifies that whenever a single database transition leaves a state in which the predicate holds with respect to some assignment to the demon parameters, and the predicate did not hold for that assignment in the pre-transition state, then <statement> is to be executed, in the post-transition state, for that assignment. A single transition may trigger several demons, and may trigger a single demon with several assignments. In this case, all such demons are to have their responses performed for all assignments, but the order in which this is to happen is non-deterministic.

On occasion, the triggering condition for a demon can be described best in terms of a transition, rather than in terms of a state, e.g., "if the price of any commodity jumps by over 7 percent, ..." To express such demons, the symbol old may be used lexically within the trigger in the same way as in a postcondition (see section 3.3).

Example

Suppose the shipping system receives periodic updates on the progress of ships at sea, in the form of latitude and longitude readings and compass headings. Suppose it also receives periodic reports on weather conditions at various locations. Finally, suppose it is capable of sending messages to ships. To support this in the specification, the domain model could include:

```
type
      weather, = (Clear,Stormy);
      heading, = integer in range [0,360];
      shiploc
end type;

relation
     WEATHERSTATUS(shiploc,weather), key is shiploc;
     SHIPPOS(ship,shiploc,heading), key is ship,shiploc;
     COORDINATES(shiploc,latitude,longitude),
         defines shiploc, key is (latitude,longitude)
end relation;

action
```

BROADCAST(*ship*,*message*), definition ...
end action;

A *demon* can specify that a warning is to be sent to any ship approaching a stormy weather area. The concept of "approaching" must of course be formally specified. The details of this are really orthogonal to the issues of data-triggered processes; the formal specification would define a many-to-many relation **APPROACHING**(*ship*,*shiploc*) and a derivation rule defining **APPROACHING** in terms of ship's latitude, longitude, and heading -- i.e., in terms of **SHIPPOS** and **COORDINATES**. The demon, which we name **STORMWARNING**, is defined by:

demon
STORMWARNING(*ship*,*shiploc*),
trigger **WEATHERSTATUS**(*shiploc*,*Stormy*) ∧ **APPROACHING**(*ship*,*shiploc*),
response **BROADCAST**(*ship*,"storm at latitude " @ **COORDINATES**(*shiploc*,☆,$)
@ " longitude " @ **COORDINATES**(*shiploc*,$,☆))
end demon;[16]

The trigger of this demon involves two relations, **APPROACHING** and **WEATHERSTATUS**, which change as the process executes. It is important to both the reliability and maintainability of a specification that this behavior be stated as a cohesive unit, rather than distributed in the various places in the process where it comes into play.

3.5 Temporal Reference

As a process executes, information is being produced and consumed. In writing a *program* to perform the process, a programmer must be concerned with the storage space required to hold this information. Programs manifest this concern by using compact or implicit representations of information, by representing only that information essential to correct execution, and, most pervasively, by releasing space used to store information that is no longer needed.[17] In a specification language, however, there is no reason to be concerned with storage space as a finite resource. As a process executes, the *current* collection of objects and associations changes, to be sure. But the history of execution and database states is conceptually well defined, in the sense that expressions and predicates can be assigned natural meanings with respect to *past* times, as well as with respect to the current state.[18] The primitive database operations destroy, delete, and update are not destructive operations but, like insert and create, alter the collection of *current* objects and/or associations.

[16] The operator @ is being used for string concatenation, converting numbers to strings when applied to numerical arguments.

[17] Programming languages include facilities, such as block structure and garbage collection, which help the programmer deal with this storage allocation problem. More importantly, as we shall see, programming languages simply do not provide certain rich constructs, whose counterparts are available in natural language, that would make the storage allocation problem too difficult for current compiler capabilities.

[18] The execution of an action that changes a previous state is not well defined, however; we leave research in this area to the producers of Star Trek and adherents to certain political ideologies.

In English, we commonly use expressions like "the President at the end of the Civil War" and "If the car was insured at the time of the accident ...". Syntactically,

<expression> at <temporal reference>
<expression> [before | after] <transition reference>

are themselves <expression>s, whose *values* are the objects described by <expression> in the referenced state. Similarly,

<predicate> at <temporal reference>
<predicate> [before | after] <transition reference>

are themselves <predicate>s. A <temporal reference> specifies a past state of the data base, and implicitly the execution history preceding and following that state.[19] It *does not* indicate a lexical point in the specification, and thus does not provide access to previous bindings of specification variables. A <transition reference> specifies a particular data transition in the process history, and thus the **before** and **after** states of that transition.

The value of temporal reference in a specification is that it enables data reference to be localized at the point where the data is needed. In a language permitting reference to *current* data only, it becomes necessary to introduce auxiliary concepts, which have no analog in the ideal world, to drag *historical* information through the execution so that it will be *current* information at the point of consumption. The existence of a global data base, changing in discrete steps and representing information in a format independent of variable bindings, provides the opportunity to incorporate temporal reference cleanly into the specification language.

Example

When filling a customer's order, a bill must be sent indicating the cost for that order. Suppose that cost is (in part) a function of the market price of the cargo *when the order was placed*, which may differ from the market price at billing time. If *cargo* and *order* refer to a particular **cargo** and **order**, respectively, then the expression

MARKETPRICE(*cargo*, *t*) before creation(*order*)

specifies the price of the cargo at the time the order was created.[20]

---

[19] This permits temporal references to be built up in expression-like fashion.

[20] The various forms for <temporal reference>s and <transition reference>s, such as creation(<expression>), have not yet been delineated. The forms appearing here are only meant to be suggestive of actual capabilities and syntax.

3.6 Process Granularity

The domain model of typed objects and associations has a "natural" processing granularity. The primitive transitions at this level are insert, delete, update, create, and destroy. The ideal process being specified, however, may have a coarser granularity. That is, some conceptually indivisible state transition in the ideal can only be described in terms of multiple primitive transitions.

Others have recognized that the granularity differences affect the checking of integrity constraints in database management systems. While it is desirable to state the integrity constraints in terms of states of the ideal process, they may be violated in the spurious intermediate states that exist as the database changes from the representation of one ideal state to another. Suppose, for example, that ship's officers in the ideal process could be reassigned on occasion to new posts, with their salaries changing as part of the reassignment activity. Suppose, furthermore, that a salary floor exists for captains. The finer grain of the data base can only represent a reassignment as a sequence of primitive operations. If the reassignment is specified by a (update post; update salary) sequence, however, the salary floor constraint may be violated temporarily when an officer is being upgraded to captain. The other order might violate the constraint when an officer was being demoted, or retired, and his salary reduced. The resolution of this problem proposed in database systems is to introduce the concept of a *transaction*, or *structured operation*, to capture the granularity of the ideal process, and to have the system guarantee the integrity of the data base only on completion of these transactions, but not within them.

The same issue must be faced in the specification language because the domain constraints and demon triggers are naturally defined with respect to states of the ideal process. But even in the absence of constraints and demons in a specification, it is important to capture the granularity of the ideal process in the specified process. The primary reason for this is the enhancement of maintainability. Adding a new constraint or demon to a specification with the wrong granularity will not yield the desired new specification. Furthermore, specification by reference to past states of a process (see section 3.5) cannot be done naturally if the specified granularity is not matched to the ideal.

Rather than indicating when (particular) constraints and demons are to be checked, the specifier should define indivisible database transitions matching the granularity in the ideal process. The resulting specification will define a process having no spurious intermediate states. The construction:

atomic <statement>$_1$; <statement>$_2$; ... <statement>$_n$ end atomic

defines an indivisible "macro" database transition as the composite of the transitions defined by the <statement>$_i$, which may range from primitive database operations to complex control structures specifying, perhaps conditionally, database transitions. The new transition defined by the block can be decomposed into the *unordered* collection of

primitive transitions so specified.[21] Since no intermediate states exist, all database conditionality in ⟨statement⟩$_i$ is based on the state existing at the start of the atomic block, and is *independent* of the partial transition specified by ⟨statement⟩$_1$, ..., ⟨statement⟩$_{i-1}$. In general, this makes it easier to specify macro transitions, since it is not necessary to give temporary names to information about the initial state that may be needed to compute the transition, but is being destroyed by the transition.

Example

Suppose the data base includes the assignment of officers to ships, and the salaries of these officers:

```
type assignment; officer; salary, = integer in range[10000, 40000];
     position, ⊃ {Captain, Firstmate}
end type;
relation
   SAL(officer, salary), defines officer;
   POST(assignment, position, ship), defines assignment;
   FILLS(assignment, officer), key is assignment, officer
end relation
```

A type **captain** could be defined as an officer filling the position *Captain* on any ship. A constraint can declare the lower bound on the salaries of **captains**.

```
type
     captain, = {[o: officer | ∃a: assignment FILLS(a,o) ∧ POST(a, Captain, $)]}
end type
constraint SAL([captain], ☆) < 15000 end constraint
```

An officer *o* could be transferred to a new assignment *a* by the action **REASSIGN**, which deletes the **FILLS** tuple indicating the previous officer filling *a*, updates the **FILLS** tuple for *o* to indicate *o*'s new assignment, and updates the **SAL** tuple for *o* to indicate a new salary, which is some function **FN** of *o*'s previous salary and the salary of the previous officer filling *a*.

```
action
   REASSIGN(o: officer, a: assignment), definition
        atomic
           delete FILLS(a, $);
           update assignment in FILLS($, o) to a;
           update salary in SAL(o, $) to FN(oldvalue, SAL(FILLS(a, ☆), ☆))
        end atomic
end action
```

The definition of **REASSIGN** is an atomic transition. The order of the three statements within that definition is immaterial. Constraints, such as the salary floor for captains, must not be violated in the state resulting from the transition. Because this is an atomic transition, the argument to FN is the salary of the previous officer filling *a*.

---

[21] This collection must satisfy certain well-formedness conditions to make sense. For example, it cannot include both insertion of a new association involving an object and destruction of that object.

It may be argued that the latter effect could be achieved even if the intermediate states did exist, either by rearranging the order of operations in **REASSIGN** or by saving the salary of the previous officer in a temporary. There are good reasons *not* to introduce such implementations of the transition into the specification. This is obvious if a somewhat more realistic situation is considered. In general, several officers may be reassigned or retired in a single transition in the ideal world. To achieve this as a sequence of state transitions would require either a sophisticated ordering of the individual reassignments or saving (potentially large amounts of) temporary data to overcome the interdependencies of the salaries. Either of these methods would obscure considerably the specification of a data transition composed of a collection of simpler transitions, each dependent only on the initial data state. The atomic construction permits a straightforward, and, thus, less error-prone, specification.

3.7 Constraints and Non-Determinism

The many forms of declarative information introduced in the preceding sections have been *constraining* in nature. They serve to categorize certain database states, or state transitions, or processing sequences as anomalous. The function of constraints in data management systems has been seen to be that of guaranteeing the integrity of the stored data [10, 11]. Any attempt to violate a constraint results in *rejection* of the database operation that would cause the violation (or, in some simple cases, a correction of the offending value).

This use of constraints has two benefits. Obviously it provides a great deal of protection to users, whether human or software, of the data. Furthermore, it opens up the potential for achieving considerable efficiency in a compilation process, through the choice of both data structures and algorithms tailored to the constrained data and constrained use thereof.

However, this use of constraints relegates them to an essentially redundant role in specification. That is, in the best of all possible worlds, all constraints would in fact be implied by the process specification and input restrictions alone.[22] In other words, if inputs to a valid implementation of our ship system were appropriate, there would be no execution that would ever "attempt" to overload a ship, and thus the capacity constraint would be implicit in the specification.

If we look at natural language, however, we find constraints playing a more active role. It is reasonable to say "choose a ship bound for Seattle and load 5000 tons of corn onto it." It "goes without saying" that the non-deterministically described ship[23] should have 5000 tons of spare capacity, and should not contain any *Oil* or *Natural Gas*,

---

[22] Although it might be very difficult to express certain constraints in terms of constraints on input.

[23] Throughout this section, non-determinism is being treated only as a way of indicating a range of alternatives, *any* of which is acceptable. An implementation of the specification is free to behave in any of the acceptable ways, or in different acceptable ways at different times, but need not cover all the alternatives or distribute its behavior among them in any particular manner.

since these cannot be combined with *Corn*. In reality, it just "goes without *re*saying." Having stated the constraints, it is unnecessary in English to refine every descriptive reference to the point where the only objects satisfying the description are those guaranteed to be "valid" in the usage context.

Likewise, it is undesirable to sprinkle predicates throughout a specification solely for the purpose of *avoiding* a conflict between the declared constraints and the specified processing. To do so would destroy the locality of the constraint declaration, not to mention the great burden it places on the specifier.[24] Rather, the constraints should affect the semantics of the remainder of the specification.

Informally, this is accomplished by viewing the alternatives available for any non-deterministic construct in the specification as being limited not only to alternatives *meeting the local requirements of the construct*, but to alternatives permitting the process to be completed without violation of any constraint.[25]

More formally, possible executions of a specification containing non-deterministic constructs can be viewed as forming a tree, with branches corresponding to alternative continuations of the process (disregarding constraints). The paths leading from the root of the tree to certain nodes may necessarily violate constraints or use anomalous statements in reaching that node. Label all such nodes **anomalous**. Then

1. Prune away all subtrees below nodes labeled **anomalous**.

2. If every leaf of a subtree is labeled **anomalous**, label the root of the subtree **anomalous**.

Repeat steps 1 and 2 until no more nodes can be labeled. Then prune away each remaining **anomalous** node and the branch linking it to its parent. If no tree remains, i.e., the root gets labeled **anomalous**, then the specification is inconsistent. Otherwise, the remaining tree represents the subset of executions actually permitted (specified), taking constraints into account. It is entirely possible, and highly likely in the envisioned usage, that locally non-deterministic constructs turn out to be entirely deterministic when constraints are considered.

Non-determinism comes into a specification in several forms. Predicates formed from patterns with *unassigned variables used freely* are frequently non-deterministic, as are pattern-based expressions. It is also possible to write expressions for "any" element of a set, to express iteration over elements of a set in a non-deterministic (including arbitrary) order, and to express a collection of distinct statements as alternative continuations of a process.

---

[24] Programmers are of course familiar with this burden, for they are generally required to "compile in" constraints when they write a program.

[25] *The use of constraint* here is to be taken, very generally, to include not only those constraints declared in the specification but also the use of anomalous statements and the universal constraints on well-formed manipulations of the data base; e.g. "thou shalt not create and destroy the same object in a primitive transition."

Local non-determinism, constrained away by global considerations, is useful for maintaining locality of information in a specification. Where true non-determinism exists in the ideal world, it is important to capture the full range of acceptable alternatives in the specification, so as not to unwittingly rule out efficient implementations by overconstraining.

Example

An action that would load a given volume of some cargo from one port onto any available ship bound for another specified port could be defined by:

```
action
   LOAD(cargo, volume, port.1, port.2),
     definition
     begin
        ∃ship PORTOFCALL(ship, port.2);
        MOVESHIP(ship, PIERPORT(☆, port.1);
        LOADSHIP(ship, cargo, volume)
     end
end action
```

The definition of LOAD is a block that assigns to its local variable *ship* a ship having *port.2* as a port of call. Then the action MOVESHIP is to be performed on that ship, positioning it as some pier in *port.1*. Finally, the cargo is actually loaded onto the ship.

MOVESHIP was defined earlier as:

```
action
   MOVESHIP(ship, pier),
        precondition INPORT(ship, PIERPORT(pier, ☆)) ,
        definition
          if MOORAGE(ship, pier) then
               comment no movement needed end comment else
               update slip in BERTH(ship, $) to SLIPS(pier, ☆)
end action
```

Considerable use has been made of the interaction between constraints and non-determinism. The predicate used to assign *ship* required only that the ship be bound for *port.2*. The precondition of MOVESHIP ensures that only ships in *port.1* can be considered, since the ship is to be moved to a pier in *port.1*. The capacity and incompatible cargo constraints, which could be violated by LOADSHIP, further restrict the choice of ships.

The pier specified in the invocation of MOVESHIP is also non-deterministically specified to be any pier in *port.1*. However, since the pier selected will be the moorage of the ship when LOADSHIP is performed, the precondition of LOADSHIP ensures that only a pier capable of handling *cargo* will be selected.

Finally, within MOVESHIP, the slip selected (in the case that the ship really needs to be moved), is constrained locally only to being any slip at the pier to which the ship is being moved. However, since ships cannot share a slip (BERTH is a 1-1 relation), only empty slips will be considered.

As a result, the constraints semantically "compile themselves" into the specification, restricting the ship, pier, and slip that must be fixed for the loading operation. A given invocation of **LOADSHIP** may be *non-deterministic*, *deterministic*, or even *anomalous*.

3.8 Anomaly Control

We give verbal recognition to the potential anomaly of a statement in English by embedding it in phrases such as "Try ...", or "..., if possible." This same capability belongs in specifications, for the alternatives are intolerable.[26] This situation is distinct from that in which a collection of equally acceptable statements is specified. Contrast "Buy four artichokes if you can; otherwise buy two pounds of peas" with "Buy either four artichokes or two pounds of peas." In a specification, the recognition of potential anomaly is dealt with by the construction:

attempt <statement>$_1$; <statement>$_2$;... <statement>$_n$ end

which has the semantics of the first <statement>$_i$ which is not anomalous.

4. CONCLUSION

This report has focused on those aspects of a formal process specification language that rely heavily on concepts developed out of Codd's original work with relational data bases. There are a number of other aspects to the language that embody the principles outlined in [2].

A good specification language, however, will not eliminate the difficulties in software development. It will simplify the mapping from "ideal world" to formal statement, and will ease the task of reflecting changes to the conception of that world in its formal counterpart. Two steps remain in the path to useful computer software.

First, a specification, as presented in this report, defines a closed world of activity responding to and altering information. It is necessary to split this closed world into a *software component* and an *environment* to adequately specify what is to be *implemented*.

Second, the mapping from a specification in a language such as this to a program of acceptable efficiency will rarely fall within the competence of existing, or even reasonably foreseeable, compilers. Human insight remains necessary. One approach is to require a programmer to relate his program to the specification in such a way that a

---

[26] This is a very difficult problem in programming. If the conditions making an activity anomalous can be tested at sufficiently low cost, a programmer will simply embed the activity in a conditional, essentially hand compiling the constraints. When the test is too expensive, or the programmer cannot even determine what an adequate test is, he must resort to unconventional control mechanisms, like error handling or backtracking.

computer, perhaps with his aid, can verify its validity [17]. Another approach is for the programmer to develop the implementation by sequentially *transforming* the specification into an acceptable program, with each step in the transformation sequence being verified (or unverifiable assumptions recorded) [3].

Not all the richness of the specification language comes from constructions that prohibit efficient automatic compilation, however. Many of the uses of patterns, type hierarchies, and typed variables appear to border on, or fall within, the capacity of current compiler technology. Data management languages are an ideal testing ground for these ideas, and we feel that their introduction would be of great benefit to users of such languages.

Formally specifying a large system cannot be made an easy task, no matter how rich or natural a language we provide. Mechanical aids to the creation of formal specifications, particularly ones permitting use of some of the power of informality in natural language [1], should help, but ultimately the creation of good specifications is an art. For, among other things, a good specification lends itself to simple maintenance. It is the insight and anticipation of the human who creates a specification that gives it this property.

REFERENCES

1. Balzer, R., N. Goldman, and D. Wile, "Informality in program specifications," *IEEE Transactions on Software Engineering* SE-4 (2), March 1978, 94-103.

2. Balzer, R., and N. Goldman, "Principles of good software specification and their implications for specification languages," in *Specification of Reliable Software*, pp. 58-67, IEEE Computer Society, 1979.

3. Balzer, R., *Transformational Implementation: An Example*, USC/Information Sciences Institute, RR-79-79, 1979.

4. Bobrow, D., and B. Raphael, "New programming languages for artificial intelligence research," *ACM Computing Surveys* 6 (3), September 1974, 153-174.

5. Bobrow, D., and T. Winograd, "An overview of KRL, a knowledge representation language," *Cognitive Science* 1 (1), January 1977, 3-46.

6. Burstall, R., and J. Goguen, "Putting theories together to make specifications," in *Fifth International Conference on Artificial Intelligence*, pp. 1045-1058, August 1977.

7. Chen, P., "The entity-relationship model -- Toward a unified view of data," *ACM Transactions on Database Systems* 1 (1), March 1976, 9-36.

8. Codd, E. F., "A relational model of data for large shared data banks," *Communications of the ACM* 13 (6), June 1970, 377-387.

9. Geurts, L., and L. Meertens, *Remarks on Abstracto*, Mathematisch Centrum, Technical Report 99, November 1978.

10. Hammer, M., and D. McLeod, "A framework for data base semantic integrity," in *Second International Conference on Software Engineering*, pp. 498-504, October 1976.

11. Hammer, M., and D. McLeod, "The semantic data model: A modelling mechanism for data base applications," in *International Conference on the Management of Data*, ACM SIGMOD, May 1978.

12. Hill, I.D., "Wouldn't it be nice if we could write computer programs in ordinary English -- or would it? " *Computer Bulletin* 16 (6), June 1972, 306-312.

13. Liskov, B., A. Snyder, R. Atkinson, and C. Schaffert, "Abstraction mechanisms in CLU," *Communications of the ACM* 20 (8), August 1977, 564-576.

14. Smith, J., and D. Smith, "Database abstractions: aggregation and generalization," *ACM Transactions on Database Systems* 2 (2), June 1977, 105-133.

15. Sowa, J., "Conceptual graphs for a database interface," *IBM J R* 20 (4), July 1976, 336-357.

16. Winograd, T., "Beyond programming languages," *Communications of the ACM* 22 (7), July 1979, 391-401.

17. Wulf, W., R. London, and M. Shaw, "An introduction to the construction and verification of Alphard programs," *IEEE Transactions on Software Engineering* SE-2 (4), December 1976, 253-265.